PATENT APPLICATION

A METHOD AND SYSTEM FOR A SECURITY MODEL FOR A COMPUTING DEVICE

Inventors: George E. Hoffman and Dianne K. Hackborn,

Filing Date: February 8, 2005

Assignee: PalmSource, Inc.

Prepared By:

Berry & Associates P.C.
9255 Sunset Blvd. Suite 810
Los Angeles, CA 90069
Phone: (310) 247-2860
Fax: (310) 247-2864

A METHOD AND SYSTEM FOR A SECURITY MODEL FOR A COMPUTING DEVICE

PRIORITY CLAIM

[0001] The present invention claims priority to U.S. Provisional Patent Application No. 60/543,108 filed on February 9, 2004, the contents of which are incorporated herein by reference.

RELATED APPLICATIONS

[0002] The present application relates to the following applications: (1) Attorney Docket No. 4001.Palm.PSI entitled "A System and Method of Format Negotiation in a Computing Device"; (2) Attorney Docket No. 4002.Palm.PSI entitled "A System and Graphics Subsystem for a Computing Device"; and (3) Attorney Docket 4004.Palm.PSI entitled "A System and Method of Managing Connections with an Available Network", each of which are filed on the same day as the present application, the contents of each Application are incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0003] The present invention relates generally to operating system software. More particularly, the invention relates to software for implementing a security model and enforcement thereof in a graphics subsystem of a computing device.

2. Introduction

[0004] Graphics subsystems and operating systems generally allow external applications and plug-in components, often authored by third-party developers (such as application programmers) and other not fully trustworthy sources, to execute in their environments.

This often raises security concerns for the native system. If the third-party component is poorly authored and thus has serious flaws, executing it within the operating system environment might cause serious or fatal error conditions. If the third-party component is malignant (such as a virus), it might intentionally steal sensitive data or perform destructive operations. Some operating systems have no way to enforce protections against poorly-written or malignant code, and so implicitly "trust" this code, when it is run, to be well-written and benign.

[0005] However, it is preferable that native systems take precautions against external faulty or malicious code from harming or infiltrating the system. Many modern systems employ the use of processes as a unit of protection; for example, third-party components like application programs might each run in their own process. In this model, code is permitted to do anything it likes within its own process, but boundaries between processes are enforced at the hardware level. Each process is provided by the system with a certain set of permissions to access services outside of that process, and code running within each process is limited to those external operations for which the process is granted permission. This use of processes as a unit of protection can be said to be nearly ubiquitous among operating systems beyond a certain level of complexity.

[0006] Implementing adequate safeguards using processes as the unit of protection requires both a policy and associated method for partitioning code and components into processes, and a policy and associated method for granting those processes permissions.

[0007] A policy and method for partitioning code into processes is important because this policy establishes the boundaries between components that can be controlled and manipulated by the system. If component A and component B are in the same process, there are no guarantees the system can provide either component with that will ensure it is protected from flaws or malice in the other. For example, code in component A could access code or data in component B, getting access to component B's sensitive data. Similarly, if component B requests and gains access to external resource C, the system has effectively given access to that resource to all code running in the same process as component B. If, however, component A and component B are running in different processes, the system can both reliably protect one's code and data from flaws or malicious code in the other, and can reliably grant a permission to one of these components without granting it to the other.

[0008] One simple, common technique for protecting the system from flawed or malicious action by components – and of protecting components from flawed or malicious action on the part of other components – is to place each newly instantiated component into a separate process of its own. For example, for each application that is to be run, a new process is created and the application is placed in it, separated both from other third-party components and from system components by secure process boundaries. However, there is significant overhead associated with each process. This includes both the memory and processing time associated with the creation and destruction of processes and the extra processing required in order to send messages across process boundaries as compared to that required for communication within a process. Thus, creating a process for each new application can be highly inefficient. Often, components will have credentials that show themselves to be trustworthy with respect to certain system operations (such as a cryptographic signature that could not realistically be duplicated by a malicious imposter). In other cases, two or more components may be regarded as trustworthy with relation to each other but not with respect to sensitive system services (for example, component A and component B might both be cryptographically signed by the same third-party publisher or vendor, but this publisher is not necessarily trusted by the system itself), and in this case it might be optimal to place both of these components into a single process.

Partitioning these trustworthy objects each into their own process can be very wasteful of system resources, and can lower the overall performance of the system.

[0009] A policy and method for granting permissions to processes is important because code running within a process can only access resources external to that process if the system grants it permission to do so. If partitioning code into processes establishes the "rule" (i.e. such and such component can only access those other components available in the same process), granting permissions to processes establishes the "exceptions" to that rule (i.e. except for these specific system services, which code running in that process has the permission to use). In many systems this is done using a central repository for storing all security knowledge, access control data, and so on. For example, if a system service needs to process a request it might ask the central authority if the requestor has permission to access that service. This design requires significant centralized knowledge of all system-wide security policies, data and policy duplication between secured components and the central authority that implements the policies, and constant communication between the central authority and objects in the system, both of which are undesirable in a dynamic, scalable, open system.

[0010] Also, in most systems, such permissions are defined and tracked in a static way with respect to processes. For example, in such a system a component would be designed to exist in its own dedicated process, and will declare security constraints for that process rather than for the component. In a system where components are partitioned more dynamically into processes, it would be preferable to define security policies at the component level. This would allow per-component information to be used to make good partitioning decisions, and would allow a network of interoperable components to define their security policies in a distributed fashion, eliminating much of the need for centralization and the overhead associated with it.

[0011] What is needed in the art is a method by which components may be partitioned into processes based on the security requirements of both the system itself and of components running in it with relation to each other, such that desired protections can be enforced while the inefficiencies associated with processes and crossing process boundaries are minimized. Furthermore, in an environment where this first method exists and components are distributed into processes based on dynamic criteria, what is required is a method allowing security policies to be defined in a distributed manner by components running in the system, both to inform process partitioning decisions and to allow for application-specific security policies to be enforced in a lightweight, scalable manner.

SUMMARY OF THE INVENTION

[0012] Additional features and advantages of the invention will be set forth in the description which follows, and in part will be obvious from the description, or may be learned by practice of the invention. The features and advantages of the invention may be realized and obtained by means of the methods, instruments and combinations particularly pointed out in the appended claims. These and other features of the present invention will become more fully apparent from the following description and appended claims, or may be learned by the practice of the invention as set forth herein.

[0013] The present invention addresses the needs in the prior art for an improved system and method for providing a security model for objects and enforcement of the model in the graphics subsystem and operating system of a computing device. The present invention comprises a system, method and computer-readable media that provide security and access control for objects and components using a capabilities model in conjunction with the use of object interfaces and enforcement of security using dynamic protection domains implemented using the process as a protection boundary around objects.

[0014] The method aspect of the invention relates to a uniform, streamlined, and flexible procedure for creating objects that contain their own security policies and are placed in protection domains when they are instantiated based on their specific security needs. The method utilizes a process boundary as the primary security or protection boundary for enforcing the security model. The security model takes advantage of the fact that most object models allow objects to have interfaces. An object's interfaces are used to determine what caller objects are capable of accessing. Thus, there is a mapping of an object's capabilities to interfaces. An object determines what a caller object is entitled to based on the investigation by the caller object and of what the caller is aware. The caller's investigation determines what other aspects of the object the caller is entitled to.

[0015] In one aspect of the present invention, a method for controlling access to an object in an operating system of a computing device is described. An interface of a target object receives a call from an external object which is aware of the existence of the interface. At the target object, it is determined whether the external object has access to other interfaces of the target object based on the first call. Access is granted to other interfaces based on this determination.

[0016] In another aspect of the present invention a method for securing an object in a computing device operating system is described. Access constraints for an object are determined. A protection domain having a security profile that corresponds to the access constraints of the first object are identified. The object is then placed in the protection domain.

BRIEF DESCRIPTION OF THE DRAWINGS

[0017] In order to describe the manner in which the above-recited and other advantages and features of the invention can be obtained, a more particular description of the invention briefly described above will be rendered by reference to specific embodiments thereof which are illustrated in the appended drawings. Understanding that these drawings depict only typical embodiments of the invention and are not therefore to be considered to be limiting of its scope, the invention will be described and explained with additional specificity and detail through the use of the accompanying drawings in which:

[0018] FIG. 1 is a diagram showing two aspects of a graphics subsystem component of an operating system for a mobile or handheld device in accordance with a preferred embodiment of the present invention;

[0019] FIG. 2 is a diagram of a display of a mobile device having user interface elements, a display server process, and a view hierarchy in accordance with a preferred embodiment of the present invention;

[0020] FIGS. 3A and 3B are diagrams of a render stream object, a sample stream of commands, and render stream branching;

[0021] FIG. 4 is a diagram of display server components and their relationships to the view hierarchy and the physical screen in accordance with a preferred embodiment of the present invention;

[0022] FIG. 5 is a diagram of a view object and selected various interfaces for communicating with other views in accordance with a preferred embodiment of the present invention;

[0023] FIG. 6 is a flow diagram of a process of mapping capabilities to interfaces in accordance with a preferred embodiment of the present invention;

[0024] FIG. 7 is a diagram illustrating objects in processes and conduits between processes implementing dynamic protection domains;

[0025] FIG. 8 is a flow diagram of a process of instantiating a new object in a dynamic protection domain; and

[0026] FIG. 9 is a block diagram of the basic components of a computing device in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0027] Various embodiments of the invention are discussed in detail below. While specific implementations are discussed, it should be understood that this is done for illustration purposes only. A person skilled in the relevant art will recognize that other components and configurations may be used without parting from the spirit and scope of the invention.

[0028] The present invention provides for systems, methods and computer-readable media that function as a graphics subsystem of an operating system intended for use primarily on mobile and handheld devices, but also executable on any computing device as described in the figures. Examples of other computing devices include notebook computers, tablets, various Internet appliances, and laptop and desktop computers. In a preferred embodiment, the graphics subsystem operates on a handheld mobile computing device such as a combination cellular phone and PDA.

[0029] FIG. 1 is a diagram showing two primary aspects of a graphics subsystem 100 of an operating system 10 in a preferred embodiment of the present invention. The two aspects are a drawing (or rendering) model aspect 102 and a transport aspect 104. Generally, a graphics subsystem is the component of an operating system that interfaces with graphics and display hardware, provides application and system software access to that hardware and to graphics-related services, and potentially multiplexes access to graphics hardware between and among multiple applications.

[0030] The drawing model aspect 102 defines a highly expressive drawing language. It allows a graphics subsystem programmer to describe an image using primitive drawing commands, including path filling and stroking, and to apply modulations of color, blending, clipping and so on. Rendering is modeled explicitly as a definition of the value of each pixel within a target area. The drawing language provides a small number of drawing primitives to modify current pixel values, including two basic types of primitives: parametric drawing operations (path definition) and raster drawing operations (blitting). More complex rendering is accomplished by compositing multiple operations. Other capabilities of the rendering model of the present invention include: arbitrary path filling, alpha blending, anti-aliasing, arbitrary two-dimensional and color-space transformations, linear color gradients, bitmap rendering with optional bilinear scaling, region-based clipping and general color modulation (from Boolean clipping to spatial color modulation). Components of the drawing model can be removed for lower-end devices. For example, components can be removed in order to not support general color modulation, anti-aliasing, or other such operations that are expensive to compute on low-powered hardware. On the other hand, the model can be configured to benefit from a full three-dimensional hardware accelerator. The drawing model aspect 102 also defines a drawing API that is used by clients to express commands in the drawing language.

[0031] Transport aspect 104 enables the transmission of drawing commands from where they are expressed by calls to the drawing API, such as within a client process, to where they are executed, typically within a server process. Transport aspect 104 addresses asynchronous operational issues. For example, it addresses the issue of how a screen or display controlled by a display server can multiplex drawing and update commands coming from different client processes and optionally execute the commands out of order if the display server determines that the resultant image would be identical.

[0032] Drawing commands originating from multiple simultaneous clients of a display server are often not strongly ordered, i.e., they can often be executed in a different order and obtain the same image as if they were executed in the order specified by clients. For example, in a preferred embodiment transport aspect 104 and drawing model aspect 102 of graphics subsystem 100 are responsible for ensuring that with drawing command groupings A, B, and C specified in the order A to B to C, wherein the commands in C overlay an area drawn into by A and B draws into an area that is not affected by either A or C, A must be executed before C but B should be permitted to draw at any time. This is very useful in situations where A, B and C originate from different client processes and the client responsible for A is slow, blocked or has crashed, and the client responsible for B is ready to continue processing. Transport aspect 104 also enables a display server to communicate with a distributed hierarchy of views, wherein each view has partial or complete ownership of certain portions of the screen, i.e., the actual pixels in those portions of the display.

[0033] FIG. 2 is a diagram illustrating a display 204 of a mobile device, the display having elements 204a, 204b, and 204c, a display server process 202, and a view hierarchy 206. Display server 202 process controls the graphical features of the user interface shown on screen 204, that is, which elements are displayed and how they are displayed. Display server 202 communicates with a view object hierarchy 206 comprised of numerous views arranged in a parent-child relationship with a root view 208 distinguishable from the other views in that it is the only view directly communicated with by the display server. Transport aspect 104 enables display server 202 to multiplex drawing commands coming from different views potentially distributed across different client processes, and execute them in the correct order or in an order the display server determines is appropriate.

[0034] Drawing commands are transported from client views to the display server responsible for graphical rendering utilizing objects that function as delivery conduits. These objects, referred to as render streams, are a feature of transport aspect 104. FIG. 3A is a diagram of a render stream object 302 and a sample stream of commands 304. Render stream 302 transports commands 304 from one or more clients (such as views) in one or more client processes to a display server in a server process. In a preferred embodiment, render stream 302 is an object instantiated by display server 202 and performs as a one-way pipe that transports commands from where they are expressed, in a client view, to where they are executed, in the display server. The client view and display server can be in different processes or can operate in the same process assuming proper security measures have been taken if necessary, or if the system components are known to be trustworthy. In a preferred embodiment, drawing commands expressed into a render stream are buffered before being transported to the display server for greater efficiency when transporting across process boundaries. There can be numerous active render streams from views to the display server.

[0035] All types of drawing commands can be transported in render stream 302. In a preferred embodiment, drawing commands, e.g., *moveto, lineto, closepath, fill <color>, stroke<color>,* and so on, resemble Postscript commands. In a preferred embodiment, render streams facilitate transmission of commands or any other data, such as pixels, modulation data, etc., in one direction. Commands that typically do not require direct responses are best suited to be transported utilizing render streams.

[0036] Drawing model aspect 102 can carry out its functions independent of any render stream. For example, if the destination for drawing commands is local, such as to a bitmap, rather than to the screen of a handheld device, the drawing model does not need to utilize a render stream (although it may use other features of transport aspect 104). In cases where the drawing model operates independent of a render stream, the same drawing model API is used. However, depending on the context, commands may be rendered immediately to a local surface, or may be transported to a display somewhere else.

[0037] In a preferred embodiment, drawing by client views occurs when they are asked by the display server to refresh the screen. This is done when a view, responsible for a certain area of pixels on the screen is invalid or 'dirty' or needs to be re-drawn for any reason, for example after some action or state change has occurred that changes the look of one or more visible components, or that adds or removes views form the hierarchy. In response to an update event, views send drawing commands to the display server so the server can change those pixels according to these commands. In a preferred embodiment, this is the only mechanism by which a view may draw to the screen.

[0038] This sequence of events encompassing the request made by the display server and the resulting drawing that occurs by clients is referred to as an update cycle. The display server initiates such a cycle by sending an update event to the root view, which will then distribute the event throughout the view hierarchy as needed to invoke views to draw, together composing the final image. Render streams are used during an update cycle to transport rendering commands from client views to the display server when the server operates in a different process or device from that of one or more of the client views, or when there are multiple systems of views operating asynchronously with respect to one another and conjoined within the same view hierarchy. These conjoined systems of views are asynchronized with respect to one another by the use of view layout root objects, as detailed below.

[0039] The graphics subsystem of the present invention allows an update to be executed serially by each view synchronously following the drawing of its predecessor in the hierarchy within a single system of views, or in parallel by multiple systems of views which operate asynchronously with respect to one another. This is enabled in part by the ability of a render stream to branch. Branching is a procedure whereby an offshoot or branched render stream is created and passed to a child view to draw at some later point that the child chooses (i.e. asynchronously with respect to the parent view performing the branching operation), while the original or parent render stream continues to be used synchronously to transport subsequent drawing commands expressed by the parent view.

[0040] FIG. 3B illustrates render stream branching. For example, a client process encompassing one or more views in hierarchy 206 may have an application user interface it wants drawn on the screen. The display server ultimately controls what is displayed on the screen and client views have the information needed to describe the desired image, and so the display server needs a render stream with which to receive drawing commands from these views. In a preferred embodiment, the display server instantiates a render stream and passes it to root view 208 along with an update event, initiating an update cycle. The commands necessary to draw the application's imagery are divided into three sequences A, B, and C, and each sequence is generated by a subset of the views comprising the application. Sequence A is generated and placed into the original render stream 306, after which render stream 308 is branched from it and given to the subset of views that generate sequence B to be used to express and transport those commands. Following this, sequence C is generated and placed into the original render stream 306. Render stream 308 is thus branched from render stream 306 at a point after commands in group A has been expressed (though perhaps buffered and not necessarily transported) but before the first command in group C has been expressed. Data transported in render stream 306 includes commands from sequence A, a token for render stream 308, and commands from sequence C. Commands in render stream 308 are from sequence B., and this render stream uses TOKEN B to identify itself when returning drawing commands to the display server.

[0041] Each act of branching creates a possibility of re-ordering by the display server. Thus, in this scenario branching has created the possibility that the commands in render stream 308 can execute concurrently or out-of-order with commands in render stream 306.

Actual parallel execution can be employed using multiple processors or hardware accelerators, or greater efficiency can be reached by re-ordering commands sent to a single graphics accelerator. The display server receives these commands from render stream 306 and 308 in parallel and decides the actual order in which the commands will be executed based on the expressed order and on the dependencies between and among the commands.

[0042] FIG. 4 is a diagram of display server components and their relationship to the view hierarchy and the physical screen. The display server 202 controls the screen and is the only entity with direct access to it. The graphics driver is divided into two components: a driver 402, which provides full access to graphics hardware registers and resides in the I/O subsystem or kernel of the host operating system, and a graphics accelerant 404, which resides in the display server 202 and provides memory-mapped access to the frame buffer and access to any graphics functions implemented by graphics acceleration hardware. The display server is comprised of the graphics accelerant, a low-level renderer 406 (also known as mini-GL), which is responsible for region filling and pixel operations, and a high-level renderer 408 which provides memory buffer management, drawing path manipulation such as stroking, rendering state management, and so on.

[0043] The display server 202 has explicit knowledge of only one view, the root view 208. From the display server perspective, root view 208 is responsible for handling events (including input and update events) for the entire screen. On simple devices the root view may in fact be the only view. If there is a single process that uses only one view and no re-ordering of views, the complexity of the design collapses into a simple code path between the display server and view hierarchy, which is highly efficient on weak hardware. On devices with more advanced hardware and user interfaces the root view distributes its responsibility for handled update and input events to a hierarchy of views.

[0044] FIG. 5 is a diagram of a basic view object 502, which has three separate interfaces named *IView* 504, *IViewParent* 506, and *IViewManager* 508. The view hierarchy operates by the interaction of parents and children within the hierarchy accessing these interfaces on one another. *IView* 504 allows manipulation of the core state and properties of a view, as well as allowing it to be added to another view as a child, such as child view 510. *IView* 504 is the interface that a parent sees on its children. Input, update, layout and other event types are propagated down the hierarchy from the display server to leaf views by each parent view making calls on its childrens' *IView* 504 interfaces, and those views in turn (either synchronously or asynchronously) making calls on their own children's *IView* 504 interfaces, and so on. The *IViewParent* 506 interface is the interface that a child view sees on its parent, such as parent view 512 and which it can use to propagate events up the hierarchy, such as invalidate events or requests for a layout to be performed. *IViewManager* 508 is the interface that a child view would use on its parent to manipulate its siblings, or that a third-party piece of code would use to add or remove children to a view.

[0045] The loose coupling of the view hierarchy is enforced in part by the hiding of certain of these interfaces from a caller that has access to certain others. For example, the default security policy for views (which can be overridden by each view as desired) is that a child view which is initially given an *IViewParent* 506 interface for its parent will not be able to convert that to an *IView* 504 interface. The view itself stores state such as the spatial 2D transformation that should apply to that view's drawing, a reference to its parent, and so on.

[0046] A capability-like security model that establishes a peer-to-peer trust network between and among a set of active objects is described in FIGS. 6 and 7. The security model allows for each object in the system to declare its own security policies, and for the trust network to be implicitly and dynamically established by the application of each policy by each individual object.

[0047] Capability security models are based on a set of simple principles. Primary among those principles is the Principle of Least Authority, which states that if an object requires access only to resource (or capability) A in order to perform its duties, it should be given access only to A, and not, for example, to arbitrary other capabilities like B and C. Additionally, if an object determines that it requires capability D, it must explicitly ask for D using another capability which permits and may grant this request.

[0048] Many object models and object-oriented programming languages allow an object defined using them to expose or publish one or more well-defined interfaces that operate on it. Each interface that an object publishes defines a set of messages that can be sent to the object or methods that can be called on it. In most interface-based object models, a client that wishes to make use of an object must first obtain a reference to an interface of that object by inspecting the object and requesting the desired interface, and then make method calls on or send messages to that interface. Such an interface is typically the smallest granularity at which access to the object is granted.

[0049] In a preferred embodiment of the present invention, security policies are defined and implemented by requiring any request to an object for a given interface to be made using an already-known well-defined interface of that object, by informing the object at the time of such request as to which interface the request was made using, by allowing each object to customize which interfaces they publish based on – among other factors – which interface the request was made using, and by mapping each object interface to its own capability-like secure communication channel when that interface is referenced or used by an object within another process or device (in other words, when a method call or message passing is performed on an interface across a secure boundary).

[0050] When an object (the "caller" object) wishes to make a request for an interface B of another object (the "callee" object), that request must be made using some interface A of the callee that the caller has already obtained through some means. This interface A could, for example, have been provided to the caller at the time it was created by the creating object, or could have been requested by the caller in an interaction with another object. The callee object will receive the request for B, and will be informed at this time that the request was made using interface A. Based on this knowledge, on whether the callee actually implements interface B, and on any other policies the callee wishes to define, the callee can decide whether or not to provide the caller with the requested interface B.

[0051] In addition, other calls made on or messages exchanged using an interface (i.e. those which do not perform interface inspection requests) can themselves return interfaces published by the called object or by any other object. In each such interaction – either explicit interface inspection or the sharing by one object with another a reference to the interface of a third – a trust network among a network of peer objects is progressively and dynamically defined. This trust network defines the range of what operations are permissible by a given object. For example, if a caller object has a reference to interface A of a callee object, it could easily obtain access (and thus should be considered as having access to) any other interfaces on the callee it can obtain by making interface requests through interface A, as well as those interfaces on other objects that the callee might have access to and might make available through A.

[0052] The decisions that each object makes as to how, when, and with whom to share its own interfaces or those of another object are up to the specific object implementation, and can be dependent on any number of factors that the object chooses to consider. In the preferred embodiment of the present invention, the primary or sole factor used to determine which interfaces of a callee can be successfully requested by a caller is simply which interface the request was made using. While other factors such as cryptographic signatures, password protection, and user verification may be used by any particular object to establish special rules based on particular situational needs, using these types of factors would require a central authority to be contacted and for certain tests to be performed on the caller to determine whether or not the caller has the necessary permissions. If an object gets a call on one of its interfaces, the object knows that the calling object has authority or permission to make that call, by virtue of having had a reference to that interface in the first place. In a preferred embodiment within any given object, the knowledge of which particular interface was used to request another is all of the information that is needed to determine which interfaces get exposed in response to that request, and thus how the trust network is built. Thus, constant permission checks with a central authority for each call to an object interface are not necessary.

[0053] Security policies can be established by dividing an object's functionality into interfaces that define categories of services grouped by what level or type of permission is required to access the service. For example, an object can have an interface that grants minimal access and can give this capability to objects that cannot necessarily be fully trusted, and can provide other interfaces that grant full access to all capabilities to other objects that can be trusted.

[0054] As shown in FIG. 5, in an exemplar of the preferred embodiment, a graphical view object has three interfaces: *IViewManager, IViewParent*, and *IViewChild*.. A hierarchy of such views operates by the interaction of parents and children within the hierarchy accessing these interfaces on one another.

[0055] The default security policy for views (which can be overridden by each view object as desired) is that *IViewManager* and *IViewParent* (if even implemented by the object) are both exposed to any request made using the *IView* interface, but that *IView* is hidden from any requests made using the other two interfaces (i.e. a request for an *IView* interfaces using either the *IViewManager* or *IViewParent* interfaces will fail), and that *IViewParent* is hidden from *IViewManager.* In other words, a caller requesting interfaces for the view using the *IView* interface will be able to obtain both *IViewParent* and *IViewManager*, a caller requesting interfaces using *IViewParent* will be able to obtain *IViewManager*, and a caller requesting interfaces through *IViewManager* will see no additional interfaces.

[0056] Each of these interfaces grants a specific set of permissions to any object obtaining a reference to it, and this default view security policy is designed around the assumption that a view essentially owns its children and can manipulate them as desired, but that a child should not in general be able to directly control or manipulate its parent. A parent has an *IView* interface for each child, and thus can control each child completely because it can access all known interfaces of the child. But because a child view is only given an *IViewParent* interface for its parent when it is added to the hierarchy, it is not able to obtain an *IView* interface for the parent, which would allow it, for example, to move or resize its parent, or to obtain yet other information such as a reference to its parent's parent. This gives the child only indirect influence over views above it in the hierarchy, as provided by the limited *IViewParent* interface. For a specific object with specific needs, a parent view could decide whether or not to provide the *IViewManager* interface in response to requests made on the *IViewParent* interface, which would in effect be granting or denying to its children the permission to gain direct access to manipulate its siblings. This permission might make sense in some situations and not in others, and so each object could choose to override the default policy as it sees fit.

[0057] Abiding by this policy allows a view hierarchy to cross many processes and propagate messages up and down the hierarchy in a uniform way without the need of special interfaces, while maintaining security.

[0058] FIG. 6 is a flow diagram of a process for an object determining security access based on an interface call in accordance with a preferred embodiment of the present invention. At step 602 a target object receives a call at one of its interfaces from an external object. For example, the target interface may have three interfaces A, B, and C, each granting varying degrees of access to the target object's functions. Interface A granting the highest degree of access and interface C, the lowest. At step 604 the target object determines whether the external object is allowed access to other interfaces by checking its own security policies. In a preferred embodiment, the target object does not check with a central authority storing security data for all objects in the system. If the external object is calling on interface A, the target object may determine that the external object has access to interfaces B an C. At step 606 the target object grants access to the external object to allow access to other interfaces as determined at step 604.

[0059] What is described above is how objects behave and define security policies regardless of what processes they exist in. For example, the caller and callee objects can be in the same or in different processes. The policy or pattern by which objects are distributed across processes does not have to be known at the time code is written or the time security policies are defined. The code is written assuming that the implicit security policies between objects as defined by interface inspection policies will be reliably enforced by the system.

[0060] However, what is described above only provides mechanisms for defining security policies. There are various classes of methods of enforcing such a model, including memory protection and language-based security. In a preferred embodiment, one such method of enforcement separates objects that must be protected from one another into separate processes.

[0061] In a preferred embodiment of the present invention, when an object in one process obtains an interface on an object in a different process (on the same device or on another), a secure conduit between those processes is established and associated with that interface reference. For each single interface on a particular object in one process, a single secure conduit exists between it and another process if that other process contains objects that hold a reference on that interface. This secure conduit is used to transmit messages and method calls on a particular interface from one process to another, and to return any result values to the caller. This is shown in FIG. 7. Thus, if an object 702 in a process 704 has an interface 706, and a reference to interface 706 is held by an object 708 in a process 710 and objects 712, 714, 716 in process 718, two secure conduits exist: one conduit 720 between process 704 and process 710 and another conduit 722 between process 718 and process 704. All calls or interactions with a given interface on a given object from the same remote process will be routed through the same secure conduit.

[0062] These secure conduits are created on-demand as needed, whenever an interface is shared with a particular remote process for the first time. The use of these secure conduits allows an object to be assured that when a request arrives from a remote process, that the caller is authorized to have made the request by virtue of having had a reference to the interface, and that this interface on which the request was made can be reliably identified. A secure conduit is only created when a process boundary exists between two communicating objects, as shown by the broken lines in FIG. 7. If an object wishes to make calls on another object in the same process, that object is referred to and called directly without the use of a secure conduit. This makes the case of local communication between objects very fast.

[0063] While the security policies established by the objects themselves (as detailed above) are always in effect, separating objects into distinct processes – and thus forcing the use of secure conduits to be used to exchange messages, data and interface references between them – is a reliable way to enforce those policies. The goal is to protect the object from malicious or faulty code, and to protect existing objects from it. But enforcement of security policies can have significant performance and memory usage implications. For example, if each object in the system were given its own process, the resulting overhead would be unacceptable. Yet, in an enforcement scheme based on processes and implemented using hardware memory protection, this is what would be needed if strict enforcement of security policies was required for every object in the system. Therefore, it is desirable to only apply enforcement of these policies when such enforcement is deemed necessary.

[0064] There are various reasons that enforcement of security policies may be deemed unnecessary. For example, if all of the components that are interacting with one another within a network of objects were authored by the same vendor, it is highly unlikely that they will be malicious with respect to one another, and thus they do not need to be protected from one another. Another example is two or more otherwise unrelated objects with identical security constraints and permissions (such as two normal third-party applications that do not deal with sensitive or private data). In both cases, clustering the set of objects into a common process would greatly reduce the memory overhead required as compared to having one process per object, and the objects themselves would be able to interact very efficiently without the creation of secure conduits to route messages between interfaces. This separation is done judiciously, for example, when it is not overridden by a blanket of trust extended to all objects published by the same vendor.

[0065] In a preferred embodiment of the present invention, a security model is enforced selectively by allowing the system to allocate objects at creation time into distinct domains referred to as dynamic protection domains. This is illustrated in FIG. 8. At step 802 a caller determines that it needs to instantiate a new object. An object instantiation facility is provided in which a caller makes a request that an object be instantiated or created as shown in step 804. The request provides that the object being instantiated be created within a dynamic protection domain which the system determines to be most appropriate. At step 806, an instantiation facility creates an object based on a variety of security factors such as security constraints and requirements declared by active objects in the system, the system's own global security policies, and any trade-offs that the system is willing to make, of security vs. performance and optimal resource utilization. At step 808 the object is placed in an existing dynamic protection domain or a new protection domain, implemented in a preferred embodiment by processes. The system may also create new dynamic protection domains if this is deemed necessary, to host the newly created object.

[0066] The object that is returned from this instantiation facility may be a reference to an interface on a remote object that was created in a remote protection domain (for example, if the newly created object needs to access sensitive data that the creator does not have permissions to access), or it might be a local object within the same process (for example, if the newly created object is cryptographically signed by a publisher that the creator has declared it trusts). Because the creator is using the interface abstraction facilities discussed above, it does not need to know or behave differently depending on whether the resulting object has been created locally or remotely. The constraints, policies and requirements used to make these decisions are implementation specific, and can vary from device to device. In a preferred embodiment of the present invention, a dynamic protection domain is implemented as a process.

[0067] Objects are clustered into protection domains based on protection policies, but are placed as "close" as possible to the caller or to a delegate that the caller designates while still obeying those policies. In other words, if an object can be created in the same process as those objects that will be interacting with it without violating security considerations, it will be.

[0068] In a preferred embodiment, in a view hierarchy, a parent view may instantiate a "plug-in" child view object without having prior knowledge of what that component does or what vendor has provided it. Based on the security constraints and policies of the parent, the parent's process, the system, and the child, the system can decide whether to load the child component into the parent's process (which would allow the greatest performance), to create a new process for the child to live in (which would provide the greatest protection), or to place the new child into some other existing process that happens to fit the child's security needs and which contains no objects holding private information (which would provide a good level of security and robustness for the parent and the system while not requiring a wholly new process to serve only a single child view object). This allows the view hierarchy, for example, to be built dynamically out of cooperating pieces of third-party code without the hierarchy configuration being known by any one component, but while still allowing protections to be maintained between those components that express a need for such protections.

[0069] Inasmuch as one embodiment of the invention described above relates to a hardware device or system, the basic components associated with a computing device are discussed below.

[0070] FIG. 9 and the related discussion are intended to provide a brief, general description of a suitable computing environment in which the invention may be implemented. Although not required, the invention has been described, at least in part, in the general context of computer-executable instructions, such as program modules, being executed by a personal computer or handheld computing device. Generally, program modules include routine programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including handheld devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, communication devices, cellular phones, tablets, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0071] With reference to FIG. 9, an exemplary system for implementing the invention includes a general purpose computing device 9, including a central processing unit (CPU) 120, a system memory 130, and a system bus 110 that couples various system components including the system memory 130 to the CPU 120. The system bus 110 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory 130 includes read only memory (ROM) 140 and random access memory (RAM) 150. A basic input/output (BIOS) contains the basic routine that helps to transfer information between elements within the personal computer 9, such as during start-up, is stored in ROM 140. The computing device 9 further includes a storage device such as a hard disk drive 160 for reading from and writing data. This storage device may be a magnetic disk drive for reading from or writing to removable magnetic disk or an optical disk drive for reading from or writing to a removable optical disk such as a CD ROM or other optical media.

The drives and the associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for the computing device 9.

Although the exemplary environment described herein employs the hard disk, the removable magnetic disk and the removable optical disk, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memory (RAM), read only memory (ROM), and the like, may also be used in the exemplary operating environment.

[0072] FIG. 9 also shows an input device 160 and an output device 170 communicating with the bus 110. The input device 160 operates as a source for multi-media data or other data and the output device 170 comprises a display, speakers or a combination of components as a destination for multi-media data. The device 170 may also represent a recording device that receives and records data from a source device 160 such as a video camcorder. A communications interface 180 may also provide communication means with the computing device 9.

[0073] As can be appreciated, the above description of hardware components is only provided as illustrative. For example, the basic components may differ between a desktop computer and a handheld or portable computing device. Those of skill in the art would understand how to modify or adjust the basic hardware components based on the particular hardware device (or group of networked computing devices) upon which the present invention is practiced.

[0074] Embodiments within the scope of the present invention may also include computer-readable media for carrying or having computer-executable instructions or data structures stored thereon. Such computer-readable media can be any available media that can be accessed by a general purpose or special purpose computer. By way of example, and not limitation, such computer-readable media can comprise RAM, ROM, EEPROM, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to carry or store desired program code means in the form of computer-executable instructions or data structures. When information is transferred or provided over a network or another communications connection (either hardwired, wireless, or combination thereof) to a computer, the computer properly views the connection as a computer-readable medium. Thus, any such connection is properly termed a computer-readable medium. Combinations of the above should also be included within the scope of the computer-readable media.

[0075] Computer-executable instructions include, for example, instructions and data which cause a general purpose computer, special purpose computer, or special purpose processing device to perform a certain function or group of functions. Computer-executable instructions also include program modules that are executed by computers in stand-alone or network environments. Generally, program modules include routines, programs, objects, components, and data structures, etc. that perform particular tasks or implement particular abstract data types. Computer-executable instructions, associated data structures, and program modules represent examples of the program code means for executing steps of the methods disclosed herein. The particular sequence of such executable instructions or associated data structures represents examples of corresponding acts for implementing the functions described in such steps.

[0076] Those of skill in the art will appreciate that other embodiments of the invention may be practiced in network computing environments with many types of computer system configurations, including personal computers, hand-held devices, multi-processor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. Embodiments may also be practiced in distributed computing environments where tasks are performed by local and remote processing devices that are linked (either by hardwired links, wireless links, or by a combination thereof) through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0077] Although the above description may contain specific details, they should not be construed as limiting the claims in any way. Other configurations of the described embodiments of the invention are part of the scope of this invention. For example, objects may use features other than interfaces of the object model to which capabilities can be matched. In another example, a protection domain can be defined by a component of the system other than a process, such as memory protection or language-based security. Accordingly, the appended claims and their legal equivalents should only define the invention, rather than any specific examples given.